FEAPI: A LOW LEVEL FEATURE EXTRACTION PLUGIN API

Alexander Lerch

zplane.development
Berlin, Germany
lerch@zplane.de

Gunnar Eisenberg

Communication Systems Group
Technical University of Berlin, Germany
eisenberg@nue.tu-berlin.de

Koen Tanghe

IPEM - Department of Musicology
Ghent University, Belgium
Koen.Tanghe@UGent.be

ABSTRACT

This paper presents FEAPI, an easy-to-use platform-independent plugin application programming interface (API) for the extraction of low level features from audio in PCM format in the context of music information retrieval software. The need for and advantages of using an open and well-defined plugin interface are outlined in this paper and an overview of the API itself and its usage is given.

1. INTRODUCTION

An increasingly important branch of music information retrieval (MIR) research deals with the extraction of musical content from audio data, in a way that it can be used e.g. for organizing or searching music databases. A first step towards that end is extracting low level features from the musical audio signal that can then serve as building blocks for constructing higher level, more semantically meaningful properties of the music.

A low level feature can thus be defined as one or more values extracted from the audio signal that can be used to describe a property of the signal but is not necessarily musically or musicologically meaningful all by itself. A few typical examples of such features are: spectral centroid, energy envelope in multiple frequency bands, MFCC's, etc.

Many applications in the MIR field require a large number of these features to be extracted from the audio signal. Different applications or research projects are using very similar or identical low level features, but they all use their own implementation of well-known and often not algorithmically complex features. A lot of redundant work is being done to "reinvent the wheel" each time. Furthermore, the integration of already implemented features in a new application is usually time-consuming even if the source code is available, and almost impossible if this is not the case.

A plugin is considered to be a library that can be linked dynamically at runtime with a previously defined interface. For the host, using a plugin means to load the library and to use the exported functions during runtime.

A commonly accepted plugin API would enable the reuse of already implemented features without additional work. Furthermore, feature extraction plugins could easily be exchanged between projects, researchers and companies, if required also in binary format to protect the intellectual property of the development party. A well-defined API can also speed up the development process, since the implementation can focus more on the algorithmic aspects and less on implementation issues like API design. The presented API tries to provide all means for developers to enable these advantages.

1.1. Overview

After a short overview of some related work, the remainder of this paper deals with the following aspects of the proposed feature extraction plugin API:

- requirements and considerations
- design and architecture
- usage
- software development kit (SDK)
- license

A summary of the most important properties of the presented API is then given and the goals for further development and maintenance are outlined. Pointers to more detailed information are provided at the end.

2. RELATED WORK

To the best of our knowledge, there is currently no widely accepted audio feature extraction plugin API in use by the MIR community.

Marsyas [1] is an audio analysis and synthesis framework with an emphasis on MIR and allows a user to extend the framework by deriving from a base class tightly integrated in the framework. As such, it does not define a true feature extraction plugin API. *Maaate!* [2] is an audio analysis toolkit that provides a plugin interface, but has a strong focus on processing MPEG sound files. Both are licensed under the GNU General Public License (GPL) [3], thus enforcing the publication of source code for all plugin or host implementations, which may prevent them from being used in a commercial context where source code distribution is not appropriate.

Besides these MIR-related APIs, several plugin APIs are commonly used in the world of audio signal processing and virtual instruments. These APIs are mainly designed for transforming an

audio stream into a new audio stream (effect processing) or for generating an audio stream in reaction to incoming MIDI events (virtual instruments). They are not easily adaptable to the demands of audio feature extraction. Examples of such APIs are *LADSPA* (Linux Audio Developer's Simple Plugin API) [4], *VST* (Virtual Studio Technology by Steinberg) [5] and *AU* (Audio Units by Apple) [6]. Some influences from these API designs can be found in the presented API.

The *VST*-SDK additionally provides an offline extension allowing audio data analysis. This extension could basically be used for feature extraction. While this has the advantage of compatibility with some already available hosts, we see the following disadvantages that make the definition of a dedicated feature extraction plugin API reasonable:

- restriction of capabilities: the offline interface is per definition not able to handle audio streams, only audio files. Furthermore it is not easy to handle or store large and complex feature sets.
- complexity of plug-in implementation: the API requires working with audio file handles, and there is a bidirectional communication between plugin and host
- non-open license: the definition of the future capabilities and extensions is under control of a company and therefore cannot easily be influenced by the requirements of researchers. Open source projects are not allowed to distribute the source files of the SDK with their code, so in the case of source code distribution every possible user has to sign an individual license agreement with the license holder

3. FEATURE EXTRACTION PLUGIN API

3.1. Requirements and Considerations

Every attempt to specify an application programming interface demands a careful consideration of the required functionality and capabilities as well as of usability and simplicity. Usually, a compromise between capabilities and ease of use has to be found since they somehow contradict each other. The (non-trivial) technical requirements for the capabilities of the feature extraction plugin API were defined as:

- support for different and possibly varying sample rates of the extracted features
- support for multiple independent instances of each plugin
- support for multidimensional features
- high probability of unique plugin identification by the host without a registration process
- support for the calculation of multiple features in one plugin, if required by the developer
- support for sufficient timing information to allow synchronization of features with different sample rates
- push-style processing of audio buffers (data source can be anything: files, live streams, ...)

The following restrictions were agreed upon to allow for simple usage and implementation of the API. They can have both technical and usability reasons:

- memory allocated internally by the plugin is never used outside the plugin, and shared memory has to be allocated by the host
- the plugin can not call host functions, i.e. the host has to poll for status requests etc.
- no file handles etc. are used in the API
- no developer-specified graphical user interface (GUI) is required to run the plugin
- only one data type (namely `float`) for inputs, outputs and parameters
- no thread safety of the API, i.e. the host has to ensure that e.g. the request for results does not interfere with a running process call

To ensure cross-platform compatibility and integration in as much programming languages as possible, the plugin interface was chosen to be defined in the programming language *C*. *C* and *C++* are commonly used by researchers and companies in an audio signal processing context and compilers are available for nearly all possible target platforms. Besides the API itself, a software development kit (SDK) providing *C++* wrapper classes is available. These classes allow easy access to a plugin from the host side as well as easy implementation of plugins by inheriting from a base class plugin on the plugin side.

3.2. Design and Architecture

Basically, the API provides two types of data inputs/outputs, called signal and parameter. Both have to be of the data type `float`. A signal, whose properties can be requested with the help of the structure depicted in Figure 1, can be both an input or a result. Input signals, which are commonly thought to be audio signals, are restricted to a constant sample rate. Parameters are used to change the plugin properties. Parameter properties are defined by means of the structure depicted in Figure 2. Both structures provide extensive plain text information, information about their range and their quantization as well as other useful data.

The input signal is passed to the plugin by simply pushing new buffers of data to the process function. The host can poll for new results at any time, taking into account the thread safety issues mentioned above.

Plugins can be distinguished by a quintuple of information: the library name itself, the plugin name string, the plugin vendor string, the vendor-specific plugin ID and the vendor-specific plugin version info.

```
typedef struct FEAPI_SignalDescription_t_tag
{
    char    acName[1024];
    char    acUnit[1024];
    char    acDescription[4096];
    float   fRangeMin;
    float   fRangeMax;
    float   fQuantizedTo;
    float   fSampleRate;
} FEAPI_SignalDescription_t;
```

Figure 1: *structure for the description of result properties*

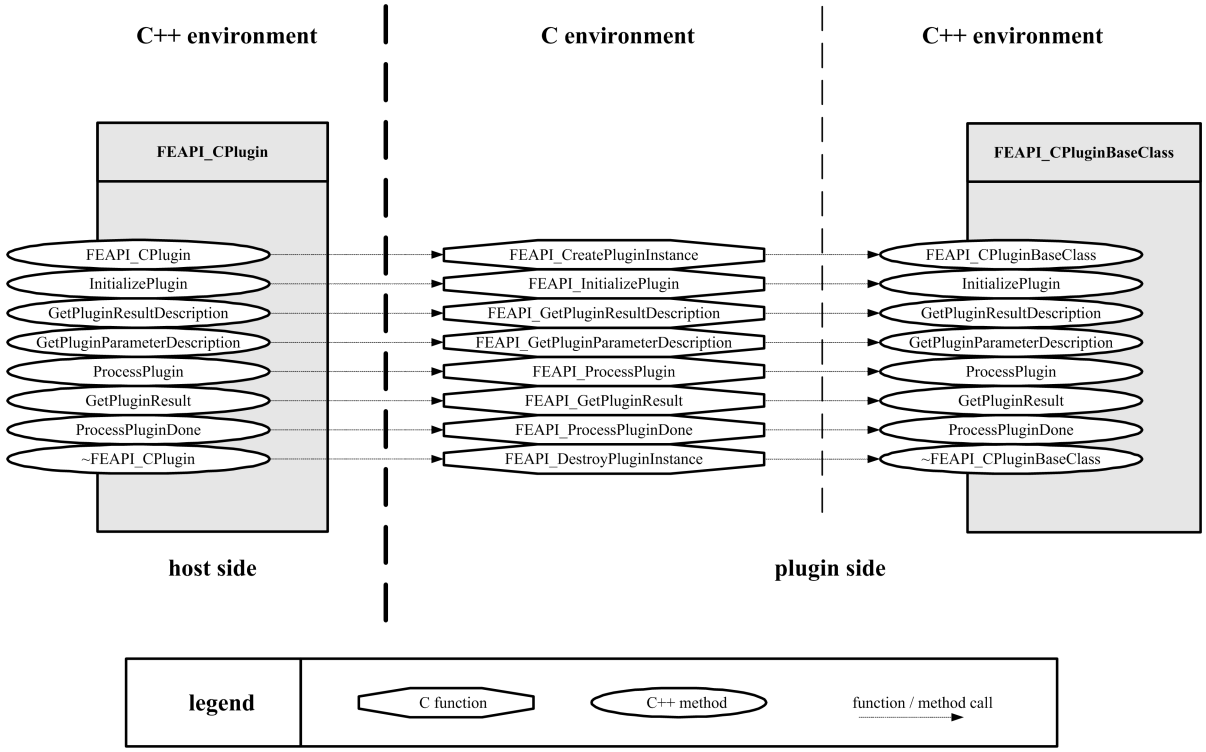

Figure 3: *SDK wrappers for the host side and the plugin side together with function / method calls*

```
typedef struct FEAPI_ParameterDescription_t_tag
{
    char    acName[1024];
    char    acUnit[1024];
    char    acDescription[4096];
    float   fRangeMin,
            fRangeMax,
            fDefaultValue;
    float   fQuantizedTo;
    int     bIsChangeableInRealTime;
} FEAPI_ParameterDescription_t;
```

Figure 2: *structure for the description of parameter properties*

3.3. Usage

Using a feature extraction plugin requires the following actions, that are also summarized in Figure 3:

`FEAPI_CreatePluginInstance()` has to be called to create a new instance of the plugin and let it do some very basic internal initialization.

The functions `FEAPI_GetPluginAPIVersion()` and `FEAPI_GetPluginCanDo()` allow the host to retrieve some plugin-specific information like API version, supported number of channels or supported sample rates. This information can be used by the host to decide whether it supports this type of plugin or not, and if it does, to call the plugin in the correct way.

The call of `FEAPI_InitializePlugin()` is required to initialize the plugin with the non-varying parameters, the input sample rate and the number of audio channels. CPU-intensive calculations necessary for internal initialization of buffers, filters, etc. should be done here too. Furthermore, some vendor-specific user data can be handed over to the plugin. If initialization fails, the plugin can not be used and the host has to destroy the plugin instance.

To retrieve information about the available plugin parameters and the calculated features, the functions `FEAPI_GetPluginResultDescription()` and `FEAPI_GetPluginParameterDescription()` can be used. Figures 1 and 2 provide some insight about the available information.

`FEAPI_ProcessPlugin()` is the function that performs the actual processing. The host simply needs to maintain a continuous stream of audio data blocks that are handed over to the plugin by calling this function. Additionally, a time stamp for the input data is passed along.

The host can then check whether a result is available or not. If so, it can check its size and the function `FEAPI_GetPluginResult()` can be called to obtain the result. A time stamp for the result is returned as well.

When no more audio data is available for processing, `FEAPI_ProcessPluginDone()` has to be called to signal the plugin that all processing is done and allow it to do some final processing based on what it has left in its internal buffers if needed.

Finally, `FEAPI_DestroyPluginInstance()` destroys the plugin instance.

Besides these functions, there are a few other ones which are

explained on the FEAPI website [7] together with more in-depth documentation on the API in general.

3.4. Software Development Kit

The API itself consists of a *C* header file containing typedef definitions of all data types that are passed through the interface and all functions which a plugin must provide. Together with the actual API, a *C++* SDK has been developed which wraps all the functions of the plugins and the API in *C++* classes. With these wrappers, a FEAPI plugin instance can be handled as a *C++* object. The specific plugin class is derived from a plugin base class (FEAPI_CPluginBaseClass). The methods provided by the base class, directly representing the functions specified by the API, are reimplemented by the specific plugin class. This structure is similar to VST and allows a very easy and fast plugin development process. The methods of the plugin object are called by *C*-style stub functions which are also provided by the SDK. These stub functions do nothing more than translating the *C* function calls into *C++* method calls.

The SDK also provides a wrapper class for host development. This class allows an easy-to-use handling of plugin libraries and the plugin instances themselves. The class (FEAPI_CPlugin) mirrors all methods which are provided by the plugin base class. With this class it appears as if the actual plugin object would have been created directly inside the host without the linking of a library.

Figure 3 shows how the SDK wraps the *C*-function calls and thus simplifies the usage of a plugin from a host. The parts programmed in *C* and programmed in *C++* are clearly separated and the border between the host's and the plugin's side is displayed. Note that only the most important functions and methods are displayed.

3.5. License

In order to have the API accepted as widely as possible, an appropriate license for the provided API source code had to be considered. The source code is licensed under a BSD-style license [8], which is a simple, permissive and widely spread license. At the same time, the BSD license is compatible with the GNU GPL [3] and the sources can be used, although under some minor restrictions, in commercial applications as well.

4. SUMMARY AND CONCLUSIONS

The presented API offers a solution for the technical requirements of low level feature extraction in an MIR context, as well as platform independence, a simple interface and an open license. The API provides a push-style interface allowing live streams as well as file streams. It supports multidimensional features to be extracted with constant as well as varying sample rates and provides sufficient information for the time synchronization of audio and features.The provided SDK, including an example plugin and host, should allow a fast learning curve. To broaden the acceptance for the usage in as many MIR-applications as possible, the establishment of a database of plugins (in source and/or binary format) for common usage is planned.

To ensure that the API and the related source code are easily available, a project has been started on SourceForge.net [7]. SourceForge provides several useful software management services like CVS, bug tracking, mailing lists etc. Source code and in-depth information about the presented API is available for download, and we encourage motivated developers in the MIR R&D field to contribute to the project by becoming an active project developer or by participating in the mailing list discussions.

5. ACKNOWLEDGEMENTS

We would like to thank Stefan Weinzierl from the Department of Communication Research and Thomas Sikora from the Communication Systems Group, both Technical University of Berlin, for their ongoing support.

Some parts of this work were done in the context of the MAMI project which is funded by the Flemish Institute for the Promotion of Scientific and Technical Research in Industry.

6. REFERENCES

[1] George Tzanetakis and Perry Cook, "MARSYAS: A Framework for Audio Analysis," *Organised Sound*, vol. 4, no. 3, 2000.

[2] Silvia Pfeiffer and Conrad Parker, "bewdy, Maaate!," in *Presentation at the Australian Linux Conference*, Sydney, January 2001.

[3] Free Software Foundation, Inc., "GNU General Public License," Available: `http://www.gnu.org/licenses/gpl.html`, last time checked: 2005 July 7th.

[4] Richard Furse, "LADSPA," Available: `http://www.ladpsa.org`, last time checked: 2005 July 7th.

[5] Steinberg AG, "Virtual Studio Technology," Available: `http://ygrabit.steinberg.de`, last time checked: 2005 July 7th.

[6] Apple Computer, Inc., "Audio Units," Available: `http://developer.apple.com/audio/audiounits.html`, last time checked: 2005 July 7th.

[7] Alexander Lerch, Gunnar Eisenberg and Koen Tanghe, "FEAPI," Available: `http://www.sf.net/projects/feapi`, last time checked: 2005 July 7th.

[8] Open Source Initiative, "BSD License," Available: `http://www.opensource.org/licenses/bsd-license.php`, last time checked: 2005 July 7th.